15th IEEE International Requirements Engineering Conference

Disambiguating the Documentation of Variability in Software Product Lines: A Separation of Concerns, Formalization and Automated Analysis

Andreas Metzger
University of Duisburg-Essen
andreas.metzger@sse.uni-due.de

Patrick Heymans
University of Namur
phe@info.fundp.ac.be

Klaus Pohl
University of Duisburg-Essen & Lero
klaus.pohl@sse.uni-due.de

Pierre-Yves Schobbens
University of Namur
pys@info.fundp.ac.be

Germain Saval
University of Namur
gsa@info.fundp.ac.be

Abstract

Feature diagrams are a popular means for documenting variability in software product line engineering. When examining feature diagrams in the literature and from industry, we observed that the same modelling concepts are used for documenting two different kinds of variability: (1) product line variability, which reflects decisions of product management on how the systems that belong to the product line should vary, and (2) software variability, which reflects the ability of the reusable product line artefacts to be customized or configured. To disambiguate the documentation of variability, we follow previous suggestions to relate orthogonal variability models (OVMs) to feature diagrams. This paper reuses an existing formalization of feature diagrams, but introduces a formalization of OVMs. Then, the relationships between the two kinds of models are formalized as well. Besides a precise definition of the languages and the links, the important benefit of this formalization is that it serves as a foundation for a tool supporting automated reasoning on variability. This tool can, e.g., analyse whether the product line artefacts are flexible enough to build all the systems that should belong to the product line.

1 Motivation

Many industry sectors face the challenge of how to satisfy the increasing demand for individualized software systems and software-intensive systems. The software product line (PL) engineering paradigm (SPLE, see [24]) has proven to empower organizations to develop a diversity of similar systems at lower cost, in shorter time, and with higher quality when compared with the development of single systems [24].

Key to SPLE is to exploit the commonalities of the systems that belong to the PL and to handle the variation (i.e., the differences) between those systems. *Commonalities* are properties and qualities that are shared by all systems of the PL [14]; e.g., all mobile phones let users make calls.

1.1 Two Kinds of Variability

In SPLE, two kinds of variability can be distinguished: Software variability and PL variability.

Software variability refers to the "ability of a software system or artefact to be efficiently extended, changed, customized or configured for use in a particular context" [31]. This kind of variability is well known from the development of single systems. As examples, an abstract Java super-class allows different specializations to be used where the super-class is used; an interface allows different implementations to be chosen.

PL variability [14, 24, 21] is specific to SPLE and describes the variation between the systems that belong to a PL in terms of properties and qualities, like features that are provided or requirements that are fulfilled. It is important to understand that defining PL variability, i.e., determining what should vary between the systems in a PL and what should not, is an explicit decision of product management (see [21, 24]). As an example, product management might have decided that the mobile phones of their PL should either offer the GSM or the UMTS protocol.

A challenging task in SPLE is to map the PL variability to software variability. This means that the reusable artefacts from which the systems of the PL are built (called the *core assets*, which constitute the *PL platform* [24]) should be constructed flexibly enough to allow for efficiently and effectively building those systems [18, 26]. This problem can be seen as an instance, in the SPLE context, of the problem of relating requirements to design. The decisions to be

1090-705X/07 $25.00 © 2007 IEEE
DOI 10.1109/RE.2007.61

made are crucial and mutually influence each other: which systems to offer as part of the PL (i.e., what the scope of the PL should be [26]), and how to design the reusable artefacts to support this scope [21].

A lack of flexibility in the reusable artefacts, or a scope that lacks awareness of the technical realizability, can severely undermine the SPLE process. At best, time-consuming and expensive changes of the reusable artefacts or the scope will be required. Therefore, it is essential to ensure that PL variability and software variability are consistent from the beginning. But since all changes cannot be anticipated, co-evolution of both variabilities over time should be facilitated too.

1.2 Documenting Variability

To make product management decisions and design decisions concerning variability explicit, thus allowing, e.g., reasoning on variability and identifying inconsistencies, this variability must be documented. To this end, Feature Diagrams (FD) are widely used in SPLE. An example is shown in the upper part of Fig. 1. Generally, a FD is a tree or a DAG[1] that serves as a compact representation of all valid combinations of its nodes, usually called *features*. A formal semantics of FDs can be devised by considering non-leaf nodes as Boolean formulae over their descendants (see Sect. 4.1).

However, a formal semantics turns out to be insufficient to dissipate all ambiguities. We studied 22 FDs from the research literature (see [23]) as well as FDs that were developed in industry (see [11]). We observed that FDs are used for different purposes. In some cases [15, 29, 9], FDs seem to be used for documenting software variability; in others [20, 21, 22, 5] they are mostly used for documenting PL variability. This had to be guessed from the context as it did not appear explicitly whether the FD is about product management decisions, about the flexibility of the PL platform, or about both.

If the two kinds of variability are not *both* explicitly documented and distinguished, assessing their consistency is difficult, especially when models become large and complex [4]. In this case, automated assistance to variability analysis becomes paramount, but is of limited use if the interpretation of the models is unclear.

1.3 Our Approach

Automated analysis of both kinds of variability is the goal we pursue in this paper. We build on previous proposals to document PL variability and software variability in separate variability models and to interrelate them (see Sect. 2.1).

[1] Directed Acyclic Graph

Starting from this separation of concerns, we formalize the syntax and semantics of those two kinds of variability models as well as the cross-links between them. We thus make all variability models amenable to automatic analysis, both separately and altogether. Based on this formalization, we devise and automate a set of checks. These deliver results that are straightforward for the stakeholders to interpret, like whether all planned systems of the PL can be realized, or whether the flexibility of the reusable artefacts is useful.

The remainder of this paper is structured as follows. After discussing related work (Sect. 2), the details on how we separate PL variability from software variability are presented (Sect. 3). The formalization of the variability models and their relationships follow in Sect. 4. Based on this formalization useful checks are defined in Sect. 5. Sect. 6 explains how tool support is achieved. The approach is applied to a comprehensive example in Sect. 7.

2 State of the Art

2.1 Separation of Concerns

Variability is a concern that affects all the core assets of the PL platform. Those assets typically comprise requirements models (e.g., use case diagrams), architectural models (like component diagrams) or test models. In the context of SPLE, these models are called *base models* [7]. Some authors have proposed extending the base model languages with constructs for documenting variability [17]. However, since variability is a cross-cutting concern that spans all base models, others have proposed to represent it separately from those models. This marks a first progress in separation of concerns.

In FORM FDs [20, 21, 22] "implemented by" links between various layers of feature refinements have been introduced that connect the top-most "capability" layer (PL variability) to the down-most "implementation technique" layer (software variability).

Other authors proposed to relate FDs to base models by specific traceability links or inclusion rules. Sinnema *et al.* propose a dedicated variability view, which shows the variability provided by the PL base models. Czarnecki *et al.* [15] propose FDs to document the valid combinations of features and to relate those diagrams to UML model templates. Similarly, Dhungana *et al.* [16] suggest relating variability models to base models. All these proposals rather focussed on separating the documentation of *software variability* from the base models.

Another series of work focussed on separately documenting *PL variability*. Those authors have favoured other notations than FDs. Bachmann *et al.* [3] are amongst the first ones to propose modelling PL variability separate from

the base models by introducing a conceptual model for variability. Becker [8] suggests an XML-based variability modelling language. John *et al.* [18] propose the concept of variability dimension to separate variability-related aspects from the base models and they suggest to document the variability of the PL in decision tables [2]. Bayer *et al.* [7] have presented a detailed meta-model of PL variability concepts, which subsumes earlier efforts. In our previous work, we have developed the OVM approach to document PL variability in graphical models (see [24, 12]) and to relate those models to the base models. Also, we have related OVMs to FDs (see [11, 13]) to eliminate specific deficits of FDs observed in the automotive industry.

Our contribution is a formal and concise approach for separating PL variability and software variability, thereby enabling automatic analysis.

2.2 Formal Semantics and Reasoning

Variability modelling languages have been studied since more than 15 years [19]. Only recently researchers have devoted their attention to the semantic foundations of those languages (see e.g., [25, 15, 1]). Their work has focused on FDs.

Our recent surveys [27, 28] showed that this research was still fragmented. To formalise, compare and automate FD languages, we introduced a systematic technique based on FFDs (Free FDs, recalled in Sect. 4.1). FFDs are a generic formalisation of the syntax and semantics of FDs. In [27, 28], most popular FD languages were (re)defined on top of FFD and compared according to formally defined criteria: expressiveness, embeddability, succinctness and complexity. A major outcome of this survey was that VFD, a specific FD dialect, obtained the best ranking on most criteria. We thus suggested to develop reasoning tools based on VFD.

Currently, the most advanced FD reasoning tools [5, 9] use off-the-shelf solvers (SAT, BDD and CSP solvers) to automate various reasoning tasks, e.g., checking satisfiability, detecting "dead" features, computing commonality, etc. (see Sect. 5). The various solvers have varying degrees of performance and coverage wrt. those tasks [10, 4].

In this paper, we follow the SAT approach since, as we will see, our analyses are largely amenable to propositional logic satisfiability problems. To be able to deal with most FD languages, we use VFD. In [28], we showed that VFD is expressively complete, and that most common FD languages can be easily and efficiently translated into it. More precisely, we will use VFD extended with textual constraints (VFD$^+$, see Sect. 4.1) as a pivot language for FD and OVM. VFD$^+$ allows for simpler mappings than VFD, and more efficient handling of models by SAT solvers [28].

As far as we know, formal reasoning has not yet been used for analysing the consistency between PL and software variability. A related but different approach was recently proposed in [15] to check that no ill-formed base models can be derived from a model template, given a correct FD configuration. The well-formedness OCL rules of the templates and the linked FD are mapped to propositional formulae (similarly to [5]). These are then fed into a SAT solver. This approach nicely complements ours, as it makes sure that the FDs are a correct abstraction of the software variability realized in the base models.

3 Separating Product Line Variability and Software Variability

To separate PL variability from software variability, we propose using OVMs to document PL variability and FDs to document software variability. We have chosen Orthogonal Variability Models (OVMs, see Sect. 2.1) as they offer a concrete, graphical syntax and tool support[2]. We decided to employ FDs, because they are a popular notation and most of the proposals for documenting software variability use them (see Sect. 2.1).

A formal definition of OVMs and FDs is given in Sect. 4. Here, we briefly recall the concepts of those models and explain how we interrelate them. Fig. 1 illustrates our approach.

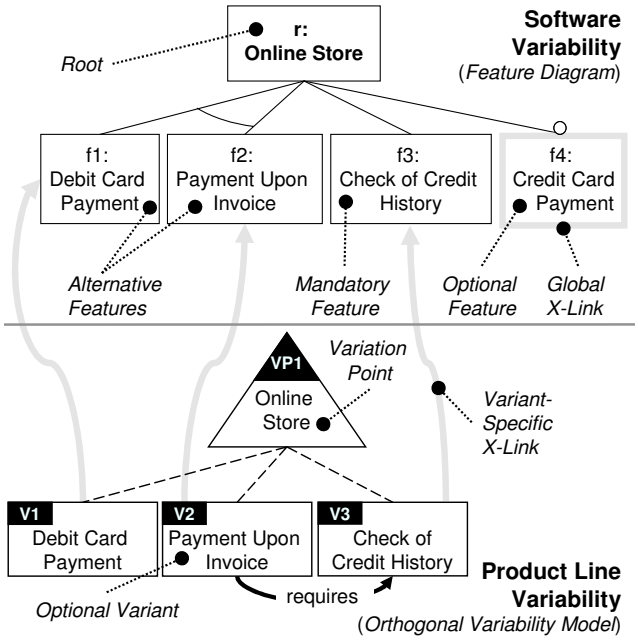

Figure 1. Separated Variability Documentation

[2] www.software-productline.com/SEGOS-VM-Tool

In an OVM [24, 12], a *variation point (VP)* documents a variable item. A *variant* documents the possible instances of a variable item and is thus related to a VP. Both VPs and variants can be either optional or mandatory (shown graphically by solid resp. dashed lines). A mandatory VP (like VP1 in Fig. 1) must always be bound, i.e., variants of that VP must always be chosen. An optional VP does not have to be bound. Mandatory variants must be chosen whenever their VP is bound. Optional variants (V1, V2 and V3 in the example) can, but do not have to be chosen. Optional variants can be grouped into *alternative choices*. The variants that can be chosen from an alternative choice are constrained by the cardinality given as $min..max$. Further constraints between variants, between VPs and between variants and VPs can be defined graphically: An *exclude constraint* specifies a mutual exclusion; e.g., if a variant excludes an optional VP, the VP may not be bound whenever the variant is chosen, and vice versa. A *requires constraint* specifies an implication; e.g., in Fig. 1, V3 must be chosen whenever V2 has been chosen.

In a FD, features are hierarchically organized, i.e., each feature can be decomposed into sub-features. In FORM diagrams [20] (like the one in Fig. 1), there can be *mandatory* features, which must be selected if their parent feature has been selected (e.g., feature f3), *optional* features (topped with a hollow circle, like f4), or alternative features (in the example, either f1 or f2 must be chosen).

Please note that in order to have a clear terminology throughout the paper, the systems that *can* be built from the PL platform will be called *products* to distinguish them from *PL members*, which are the systems that have been defined by product management *to be* built. This implies that products are specified by the FDs and PL members by the OVMs.

To express how PL variability is realized by software variability, variants in an OVM are related to features in the FD through X-links (cross-model links[3]). Whenever a variant is chosen for a PL member, all features that are x-linked to this variant must be contained in the product that is built from the PL platform. Additionally, whenever an x-linked feature is contained in a product, at least one of its x-linked variants must have been chosen for the PL member, i.e., features related to the OVM via X-links must be *justified* by variants. This reflects a typical concern in SPLE practice not to deliver more features in a PL member than actually paid for by the customer. In addition to variant-specific X-links, global X-links can be defined which imply that the feature is always included (e.g., see f4 in Fig. 1).

When analysing the variability of the on-line store PL, two inconsistencies between PL variability and software variability have been uncovered:

- Realizing features f1 and f2 as alternatives conflicts with PL variability, because variants V1 and V2 can be combined without any constraint.
- Feature f3 is realized as a commonality in the PL platform. However, f3 should only be contained in a product when V3 is chosen.

In general, those kinds of inconsistencies can be resolved by modifying the PL platform so as to support the desired PL variability, or by altering the set of PL members, i.e., by changing the scope of the PL.

4 Formalization

We start this section by defining VFD$^+$, our pivot language. Then, translations of FD, OVM and X-links to VFD$^+$ are explained. The reasoning possibilities on the resulting VFD$^+$ are explained in Sect. 5.

4.1 Formalizing Feature Diagrams

VFD$^+$ is based on FFD. FFD [28, 27] is a parametric construct designed to define the syntax and semantics of FODA-inspired FD languages in a generic way. Its *abstract syntax* ($\mathcal{L}_{FFD}$) has 4 parameters reflecting the 4 syntactic 'variation points' we observed among languages: (1) the *graph type* (TREE or DAG), (2) the types of Boolean operators used (and, xor, or, $card$,...), (3) what kind of additional *graphical constraint types* are used[4], and (4) the *additional textual constraint language*[5]. In [28, 27], we have defined most common FD dialects on top of FFD.

The abstract syntax of our pivot language is $\mathcal{L}_{VFD^+} = \mathcal{L}_{FFD}(\mathsf{DAG}, card, \emptyset, \mathbb{B}(P_F))$. Thus, VFD$^+$s are DAGs (i.e. can be trees, but do not have to). They use only one type of Boolean operator viz. $card$, which denotes the set of operators of the form $card_s[i..j]$. These were suggested in [25] and subsume all commonly used operators (or, and, xor and opt[6]). $card_s[i..j](n_1, \ldots, n_s)$ returns TRUE iff at least i and at most j of its s arguments are TRUE. Finally, all additional constraints in VFD$^+$ are expressed as propositional formulae over P_F ($\mathbb{B}(P_F)$). P_F is the set of primitive nodes, i.e. all user-relevant nodes in the FD (see below). VFD$^+$ is expressively complete, and the aforementioned FD languages can be easily and efficiently translated into it [28]. Def. 4.1 formally presents the abstract syntax of VFD$^+$. We insist that VFD$^+$ is not meant as a user language, but only as a formal "back-end" language used to

[3] In this paper we do not mean to be prescriptive about the concrete syntax of such links.

[4] Typically, cross-cutting node-to-node links labelled `requires` and `mutex` similar to those in OVM are used.

[5] Several languages use *Composition Rules* [19] to express `requires` and `mutex` on nodes in textual form; some do not use any textual language; and some use the whole power of propositional logic [6].

[6] *opt* is the operator that always returns TRUE.

define semantics and automating reasoning. A visual illustration of a VFD$^+$ is given in the lower part of Fig. 2, but is only meant as an illustration of the transformations.

Definition 4.1 (VFD^+)**.** *A VFD^+ $F \in \mathcal{L}_{VFD^+}$ is a tuple $(N_F, P_F, r_F, \lambda_F, DE_F, \Phi_F)$ where:*

- *N_F is the set of* nodes *among which r_F is the* root*; in FDs, nodes are usually meant to represent features;*
- *$P_F \subseteq N_F$ is the set of* primitive nodes*, i.e. the set of nodes that the modeller considers relevant. Hence,* primitive *nodes and* leaf *nodes are different concepts (although the former usually includes the latter) because P_F can include intermediate nodes deemed relevant by the modeller. $N_F \setminus P_F$ typically contains additional nodes introduced when mapping concrete or other abstract languages to VFD^+ (e.g. see Sect. 4.2.1 and Fig. 2).*
- *$\lambda_F : N_F \rightarrow card$ labels each node with an operator;*
- *$DE_F \subseteq N_F \times N_F$ is the set of* decomposition edges*; if $(n_1, n_2) \in DE_F$, n_1 is called n_2's* parent*, and n_2 is one of n_1's* sons*;*
- *$\Phi_F \subseteq \mathbb{B}(P_F)$ are the additional textual constraints.*

In [28], we also give some additional well-formedness rules for FFDs. All except those related to graphical constraints also apply here, but are omitted for brevity. We just note that leaf nodes are labelled with $card_0[0..0]$ as a convention.

The *semantic domain* ($\mathcal{S}_{FFD}$) of FFD is shared by all languages defined on top of FFD. $\mathcal{S}_{FFD}$ is recalled in Def. 4.2:

Definition 4.2 (Configuration, Product, Product Set[7])**.** *(1) A* configuration *is a set of nodes, i.e., any element of $\mathcal{P}N_F$. (2) A* product *is set of primitive nodes, i.e., any element of $\mathcal{P}P_F$. (3) A* product set *is any element of $\mathcal{P}\mathcal{P}P_F$.*

The *semantic function* $[\![\bullet]\!]_{VFD^+} : \mathcal{S}_{VFD^+} \rightarrow \mathcal{P}\mathcal{P}P_F$ assigns a product set to every diagram. It is formalized in Def. 4.3 and 4.4, and is just a special case of $[\![\bullet]\!]_{FFD}$ as defined in [28].

Definition 4.3 (Semantic function)**.** *The semantics of a VFD^+ F is a product set consisting of the valid products of F, i.e. its valid configurations (Def. 4.4) restricted to primitive nodes: $[\![F]\!]_{VFD^+} = \{c' \mid c \vDash F \wedge c' = c \bigcap P_F\}$. (When the type of diagram is obvious, $[\![F]\!]_{VFD^+}$ is abridged to $[\![F]\!]$.)*

Definition 4.4 (Valid configuration)**.** *A configuration $c \in \mathcal{P}N_F$ is* valid *for a $F \in \mathcal{L}_{VFD^+}$, noted $c \vDash F$, iff:*

1. *The root is in: $r_F \in c$*
2. *The meaning of nodes is satisfied: if a node $n \in c$ has sons $n_1, \ldots, n_s$ and $\lambda_F(n) = card_s[i..j]$, then $card_s[i..j](n_1 \in c, \ldots, n_s \in c)$ must evaluate to* TRUE*.*
3. *The configuration must satisfy all textual constraints: $\forall \phi \in \Phi_F, c \vDash \phi$, where $c \vDash \phi$ means that we replace each node name n in ϕ by the truth value of $n \in c$, evaluate ϕ and get* TRUE*. For instance, if ϕ is the $\mathbb{B}(P_F)$ constraint $\mathsf{f}_1 \Rightarrow \mathsf{f}_2$, then $c \vDash \phi$ when $(\mathsf{f}_1 \in c) \Rightarrow (\mathsf{f}_2 \in c)$ evaluates to* TRUE
4. *If s ($\neq r_F$) is in the configuration, one of its parents n, called its* justification*, must be too: $\forall s \in N_F \cdot s \in c \wedge s \neq r_F \cdot \exists n \in N_F \cdot n \in c \wedge (n, s) \in DE_F$.*

4.2 From Diagrams to VFD$^+$

As inputs to the translation process, we have a FD (F), an OVM (Ω) and X-links (χ) between them.

F may have been written using one of a variety of FD languages, but we do not have to assume a specific one is used: a straightforward implementation of the translations defined in [28] lets us map any of the aforementioned FD languages to VFD$^+$. So, we only need to make Ω and χ amenable to formal reasoning by translating them to VDF$^+$ as well.

4.2.1 From OVM to VFD$^+$

Currently, OVM's abstract syntax only exists as a metamodel (see [24]). Here, we define a formal version of it and, most importantly, describe a translation from OVM to VFD$^+$. This gives OVM a formal semantics and makes it suitable for automated reasoning.

We consider an OVM Ω to be a tuple of the form $(VP, V, VG, Parent, Min, Max, Opt, Req, Excl)$ where:

- $VP(\neq \emptyset)$ is the set of variation points;
- $V(\neq \emptyset)$ is the set of variants; $VP \cap V = \emptyset$;
- $VG(\neq \emptyset) \subset \mathcal{P}(V)$ is the set of variant groups; VG partitions V;
- $Parent : V \cup VG \rightarrow VP$ returns the parent VP under which a V (resp. VG) appears;
- $Min : VG \rightarrow \mathbb{N}$ and $Max : VG \rightarrow \mathbb{N} \cup \{*\}$ return cardinality of a given VG;
- $Opt : VP \rightarrow \mathbb{B}$ denotes VP optionality;
- $Req \subseteq (V \times V) \cup (V \times VP) \cup (VP \times VP)$ encodes the V-V, V-VP and VP-VP `requires` links; $Excl$ is similar for `excludes` links.

Additional well-formedness rules are:

- All VPs are parent of at least one V: $\forall vp \in VP \cdot \exists vg \in VG \cdot Parent(vg) = vp$ and $\forall vg \in VG \cdot vg \neq \emptyset$.

[7] In [28], *Product Set* was termed *Product Line*. We have changed the term here in order to avoid confusing PL and software variability.

- A VG and its Vs have the same parent: $\forall vg \in VG, v \in V \cdot v \in vg \Rightarrow Parent(v) = Parent(vg)$.
- Cardinalities must be well-formed: $\forall vg \in VG \cdot 0 \leq Min(vg) \leq Max(vg) \wedge 1 \leq Max(vg) \leq \sharp vg$.

Such an Ω can be transformed into a VFD$^+$, say $O = (N_O, P_O, r_O, \lambda_O, DE_O, \Phi_O)$, by applying the linear transformation described in Algorithm 1 (see Appendix A): Vs become primitive leaf nodes (see subroutine in Algorithm 2, Appendix A) whereas VPs become primitive non-leaf nodes and VGs become non-primitive nodes. Hence, the semantic domain of an OVM is considered the set of sets of combinations of Vs and VPs[8], i.e. $\mathcal{PP}(V \cup VP) = \mathcal{PP}(P_O)$. The semantic function of VFD$^+$ (and also that of FFD) appears to be fully adequate. In particular, the justification rule (see Definition 4.4) turns out to be useful here too, in order to prevent Vs to be chosen when their VP is not bound. Finally, at the end of Algorithm 1, we note that `requires` and `excludes` constraints are translated to textual constraints in Φ_O. An illustration of the transformation is given in Fig. 2.

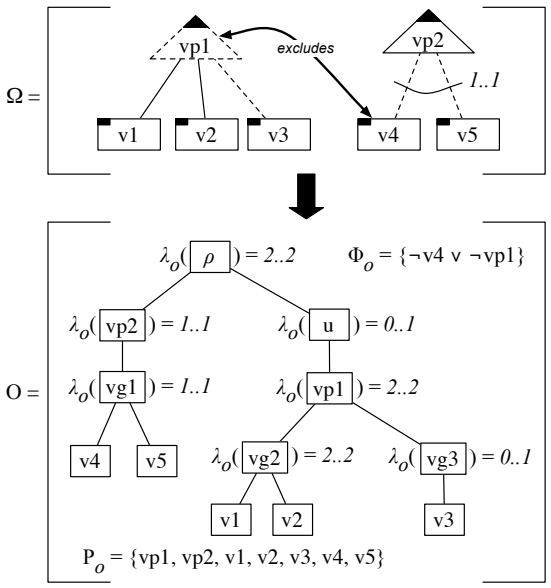

Figure 2. OVM to VFD$^+$ example

4.2.2 From X-Links to VFD$^+$

We now have two VFD$^+$'s: F and O, which embed the semantics of the FD and the OVM Ω respectively, but we also need a fomal representation of the X-links between them.

In Sect. 3, we considered two kinds of X-links:

- The first was used to include a feature in the software whenever a variant was chosen. It was shown as an arrow from a variant to a feature in Fig. 1. For each feature f that is the target of one of more such arrows coming from variants v_1 to v_n, the following must hold: $\mathsf{f} \Leftrightarrow \mathsf{v}_1 \vee \mathsf{v}_2 \vee \ldots \vee \mathsf{v}_n$.

- The second was used to include a feature in all PL members. It was represented by colouring the border of the feature. For each such feature, this simply requires that the atomic formula f holds.

Although we believe those two types of X-links are among the most frequent, we do not think we should limit X-links to those two types; e.g., exclusion constraints could be useful too. Hence, to remain general, we consider the X-links (χ) to be a set of Boolean formulae on variants and VPs (from Ω) and primitive features (from F), i.e. $\chi \subset \mathbb{B}(P_O \cup P_F)$.

We formally relate F, O and χ by applying Algorithm 3 (see appendix A). This creates a global VFD$^+$, G, that merges F and O under a common root and-node ($card_2[2..2]$ in this case), and adds χ as textual constraints in Φ_G.

The semantic domain of the merged VFD$^+$, G, is $\mathcal{PP}(P_G) = \mathcal{PP}(P_O \cup P_F) = \mathcal{PP}(V \cup VP \cup P_F)$.

5 Reasoning

Once we have VFD$^+$s F, O and G, we have three models with well defined formal semantics. Furthermore, as these VFD$^+$s were produced from original models that clearly separated the two kinds of variability (PL and software variability) and related them formally, we can now formally characterize checks that are of central interest to the PL stakeholders. And, we can automate the computation of those checks.

To start, we recall some basic formalisable checks about VFD$^+$. Several of them can be performed on other types of languages as well. Benavides *et al.* informally review those (and other) checks in [10] in relation to FDs. We select the fundamental ones and add formal definitions.

Satisfiability ($[\![x]\!] \neq \emptyset$) is a basic property. It checks whether a VFD$^+$ is consistent, i.e. whether it allows for at least one configuration of primitive nodes (cf. Definition 4.2). *Membership*, a.k.a. *product checking*, is the verification that a given configuration is accepted by a VFD$^+$, i.e. $p \in [\![x]\!]$. One can also compute the *commonality*, i.e. the set of primitive nodes that appear in all configurations ($\bigcap [\![x]\!]$) or, conversely, the presence of *dead nodes*, i.e. those that do not appear in any configuration ($P_x \setminus \bigcup [\![x]\!]$).

When applied to a FD where PL and software variability are mixed, it is not clear what those checks mean exactly.

[8] As clarified in Sect. 3, we call those combinations PL members to distinguish them from feature combinations, called products (cf. Definition 4.2)

However, in our case, formality meets intuition: $[\![F]\!]$ denotes the set of products (in terms of primitive feature combinations) *that the software platform allows to build*, while $[\![O]\!]$ is the set of PL members (expressed in terms of V and VP combinations) *that the PL management decides to offer*; $[\![G]\!]$ denotes the set of *realizable* PL members (in terms of primitive features, V and VP combinations).

The soundness of these models can first be tested with basic checks: are they satisfiable separately? Do they have dead nodes (a symptom of over-constrained models)? Commonality (a symptom of under-constrained models)? These are well known checks for which popular techniques are discussed in [10, 4].

With our approach, more advanced checks can be performed[9]:

C1 The crucial one is *PL realizability*: Are there non-realizable PL members offered in the PL? A PL member $po \in [\![O]\!]$ is *realizable* if it has a linked realization: $po \in [\![G]\!]|_{P_O}$. Un-realizable PL members are given by the set $[\![O]\!] \setminus [\![G]\!]|_{P_O}$, which we hope to be empty. Non-realizable PL members can, for instance, occur due to newly discovered exclusion constraints in the platform, coming from newly detected feature interactions. In Fig. 1, $\{\mathsf{V1}, \mathsf{V2}\}$ is detected as not being realizable.

C2 Are there identical feature combinations (i.e., products) that realize two distinct PL members? This shows a risk of *internal competition* in the PL, since customers would take the cheapest PL member. This is expressed as $(po_1 \cup pf) \in [\![G]\!] \wedge (po_2 \cup pf) \in [\![G]\!] \wedge (po_1 \neq po_2)$.

C3 Is there some *commonality between realizable* PL members? The detected commonalities $\bigcap [\![G]\!]|_{P_O} \setminus \bigcap [\![O]\!]$ help to pinpoint the problem when PL members are not realizable. For example, restricting the scope by making an optional variant or VP in this set mandatory, could help make O fully realizable.

C4 Are there some *dead variants/features in the realizable* PL members? I.e., $P_O \setminus \bigcup [\![G]\!]|_{P_O}$. This is dual to **C3**: it points optional variants or VPs that should be removed from the scope or realized.

Symmetrical questions can be formulated concerning the PL platform:

C1' We call a product *useful* if it is a possible realization of a PL member. This is stated as $pf \in [\![G]\!]|_{P_F}$. The list of non-useful products $[\![F]\!] \setminus [\![G]\!]|_{P_F}$ is a symptom of unused flexibility of the PL platform. It can be on purpose, e.g. justified by future PL extensions, or to avoid coupling among software components.

C2' A PL member can be realized by several products, another symptom of unused flexibility of the platform: $(po \cup pf_1) \in [\![G]\!] \wedge (po \cup pf_2) \in [\![G]\!] \wedge (pf_1 \neq pf_2)$. For such PL members po, the list of realizing products $\{pf | (po \cup pf) \in [\![G]\!]\}$ is the basis for selecting a realization.

C3' We can further ask for the *commonality of useful* products: $\bigcap [\![G]\!]|_{P_F} \setminus \bigcap [\![F]\!]$. These features can be made mandatory without harming the PL scope.

C4' Similarly, *dead nodes of useful* products ($P_F \setminus \bigcup [\![G]\!]|_{P_F}$) can be removed without harming the PL scope.

6 Tool Support

We are currently developing comprehensive tool support for variability management, including a graphical front-end supporting the editing and debugging of OVMs, FDs and X-links. Here, we can only focus on the reasoning capabilities offered by the tool.

6.1 VFD$^+$ to SAT

To automate checks C1 to C4', we map the checks to Boolean satisfiability problems (SAT). Our prototype uses the state-of-the-art SAT solver library SAT4J[10], also used in [5, 9]. This SAT solver requests a Boolean formula in conjunctive normal form (CNF) and, in return, delivers all variable assignments that evaluate the input formula to true. If no such assignment exists, the formula is unsatisfiable. In [6], Batory presents how to map a FD dialect $\mathcal{L}_{FFD}(\mathsf{TREE}, and \cup xor \cup or \cup \{opt_1\}, \emptyset, \mathbb{B}(N_F))$ to CNF. The mapping of VFD$^+$ is more complex since we generalise to DAGs and $card$ operators.

To go from VFD$^+$ to CNF, we need to translate both the graph and the textual constraints (Φ_x). Just as Batory's, our Φ_x consists of Boolean formulae for which we simply reuse a standard conversion to CNF. For the graph, we devise translation rules to apply to all nodes in N_x (see Table 1). For each node g with parents $f_1, \ldots, f_m$ and sons $h_1, \ldots, h_n$, a specific formula is generated depending on the operator labelling g[11]. The functions $GT^{n,i}_{SEQ}()$ and $LT^{n,j}_{SEQ}()$ are defined in [30]. They return optimally encoded formulae expressing that at least i (resp. at most j) out of n propositions are true.

Additionally, for each non-root node ($g \in N_x \setminus r_x$) we generate $\neg g \vee f_1 \vee f_2 \vee \ldots \vee f_m$ (justification rule). For the root ($g = r_x$), we just add the formula g.

[9] In the sequel, we use the notation $|_X$ to denote projections of the semantics on a particular set of nodes (typically P_O or P_F): $S|_X = \{y \cap X \mid y \in S\}$.

[10] http://www.sat4j.org

[11] Some formulae in Table 1 are not yet in CNF. This is for readability reasons. They are converted to CNF with a standard CNF conversion algorithm.

$If \lambda_x(g) = \ldots$	$\ldots$, then generate formula:
$card_n[0..0]$ $(n \geq 1)$	$\bigwedge_{i=1..n}(\neg g \vee \neg h_i)$
$card_n[0..n]$	no output generated
$card_n[1..n]$ $(n \geq 1)$	$\neg g \vee h_1 \vee h_2 \vee \ldots \vee h_n$
$card_n[n..n]$ $(n \geq 2)$	$\bigwedge_{i=1..n}(\neg g \vee h_i)$
$card_n[i..j]$ $(n \geq 2, 1 \leq i \leq j < n)$	$\neg g \vee (GT^{n,i}_{SEQ}(h_1, \ldots, h_n) \wedge LT^{n,j}_{SEQ}(h_1, \ldots, h_n))$

Table 1. Translation from VFD$^+$ to $\mathbb{B}(N_F)$

This is how we generate $CNF(x)$, from any VFD$^+$ x. When fed with $CNF(x)$, the SAT solver computes an answer for $[\![x]\!] \neq \emptyset$ and if the answer is yes, SAT4j can enumerate all configurations in $[\![x]\!]$. This can be applied to F, O and G. We now discuss checks **C1** to **C4'**.

6.2 Solving problems

For maximizing the performance of the reasoner, our goal is to have most of the computations performed *within* the SAT solvers. The main challenge is thus to reduce the checks from Sect. 5 to SAT.

C1 PL realizability cannot be easily reduced to SAT:
- (a) But we can enumerate all PL members po offered in O (using a SAT enumerator), and for each we check[12] $\ulcorner po \wedge \chi \wedge F \urcorner$ using a SAT solver, where $\ulcorner po \urcorner$ is the CNF formula $\bigwedge_{o \in po} o \wedge \bigwedge_{o \in P_O \setminus po} \neg o$. However, the number of offered PL members could be exponential.
- (b) A cheaper solution is possible when the X-links, χ, determine a single product per PL member. This condition is checked by the non-satisfiability of $\ulcorner O \wedge \chi \wedge \chi_1 \wedge (f \neq f_1) \urcorner$, where χ_1 is χ in which the symbols $f \in N_F$ have been replaced by fresh symbols f_1, and $(f \neq f_1)$ stands for $\neg(\bigwedge_{f \in P_F} f \equiv f_1)$. Following [15], we can then check the satisfiability of $\ulcorner \neg(O \wedge \chi \Rightarrow F) \urcorner$.

C2 Cases of *internal competition* will be given by a SAT solver on the formula $\ulcorner F \wedge (\chi \wedge O) \wedge (\chi' \wedge O') \wedge (o \neq o') \urcorner$, where χ', O' are χ, O where all symbols in $o \in P_O$ have been renamed to a fresh symbol o', and $(o \neq o')$ is an abbreviation for $\neg(\bigwedge_{o \in P_O} o \equiv o')$.

C3 The commonality of realizable products can be computed by intersecting them.

C4 Similarly, the dead nodes are computed as the intersection of the complement of the realizable products.

Checks **C1'** to **C4'** are treated symmetrically.

[12] $\ulcorner \phi \urcorner$ denotes the formula ϕ where the VFD$^+$s occurring in it are replaced by their corresponding Boolean formulae, and where the overall formula has been put in CNF, if possible.

7 Evaluation

As a first step toward a comprehensive evaluation, we analysed the variability of a PBX (Private Branch Exchange) PL inspired from [22]. Fig. 3 shows the separated documentation of variability for the PBX PL. It should be noted that in contrast to the simple example from Sect. 3, this OVM also includes variants that represent product types or categories (cf. [13]).

We applied our transformations to the PBX models to generate VFD$^+$s and then the CNFs $\ulcorner O \urcorner$, $\ulcorner F \urcorner$ and $\ulcorner G \urcorner$. Checking the individual models did not reveal any problems. All were satisfiable and had no dead nodes. However, when testing for **C1**, 9 non-realizable PL members were discovered. In more detail, the following defects have been located in the models:

- The PL member $\{$V1.3$\}$ violates the f24 requires f3 constraint. This defect can be resolved by introducing a requires dependency in the OVM from V1.3 to VP3, thus reducing the PL scope.
- The PL member $\{$V1.2$\}$ violates the f21 requires f2 constraint. Similarly to above, a requires constraint from both V1.2 and V1.3 to VP2 could be introduced to resolve this issue.
- PL member $\{$V1.2, V2.2$\}$ violates f8 being mandatory, as f8 should not be in the PL member when V2.1 is not bound. As a solution, the PL platform could be modified such that f9, f10 and f8 can be offered with cardinality 1..3.

It took less than a second to a low-end laptop PC to compute the results of all 8 checks **C1**-**C4'**[13].

8 Conclusion and Perspectives

In this paper, we proposed a model-based technique to unambiguously document and automatically analyse variability in software product lines. The approach is characterized by a clear separation between product line variability

[13] Detailed results are available at www.sse.uni-due.de/paper/PBX-Analysis-Results.pdf

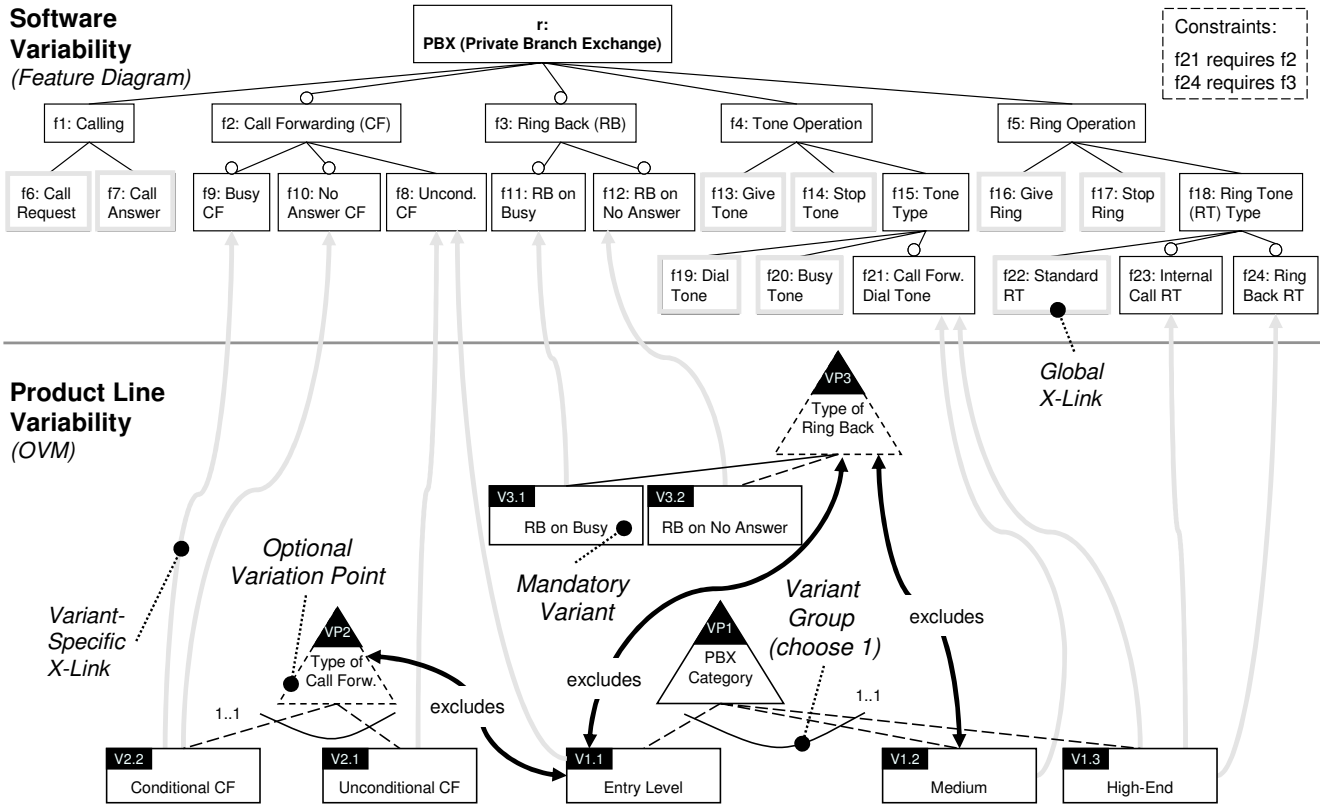

Figure 3. Separated Documentation of Variability of a PBX

and software variability, and by explicitly modelling of each of them as well as their interrelationships. Disambiguation is further supported by formally defined modelling languages and cross-model links. Separation and formality allow for the automation of crucial consistency checks during software product line engineering, both at early stages of development as well as during product line evolution. A crucial check, for instance, is realizability, i.e., whether the PL platform is flexible enough for realizing all PL members as envisioned by product management.

There are many directions in which this work should be extended. First, our list of consistency checks is by no means exhaustive. We will complement it with additional automated checks deemed relevant to practitioners. While some optimizations are already presented here, improving the performance of the checks is crucial too. This quest for efficiency will have to be driven by benchmarks and the study of complexity of the algorithms. Also, consistency between variability models and other product line artefacts (e.g., requirements, UML diagrams, components or test cases) is an important and challenging target for automation. Finally, an extensive industrial application and evaluation is on our agenda.

Acknowledgements

The first two authors acknowledge the generous support by Lero and its staff during their research visit to Limerick. We thank Kim Lauenroth, Ernst Sikora and David Benavides for fruitful discussions, and J-C Trigaux for help in tool development. Parts of this work have been sponsored by DFG under grant PO 607/1-1 PRIME, SFI under the CSET grant 03/CE2/I303_1, EU under FP6 grant InterOP 508011, DGTRE under grant PLENTY and BELSPO under grant MoVES.

References

[1] T. Asikainen, T. Mannisto, and T. Soininen. A Unified Conceptual Foundation for Feature Modelling. In *Proceedings of the 10th International Software Product Line Conference*, pages 31–40, 2006.

[2] C. Atkinson, J. Bayer, C. Bunse, et al. *Component-based Product Line Engineering with UML.* Addison Wesley, 2001.

[3] F. Bachmann, M. Goedicke, J. C. S. do Prado Leite, R. L. Nord, K. Pohl, B. Ramesh, and A. Vilbig. A meta-model for representing variability in product family development. In *Software Product-Family Engineering, 5th Int'l Workshop,*

PFE 2003, Siena, Italy, November 4-6, 2003, Revised Papers, volume 3014 of *LNCS*, pages 66–80. Springer, 2003.

[4] D. Batory, D. Benavides, and A. Ruiz-Cortes. Automated analysis of feature models: challenges ahead. *Commun. ACM*, 49(12):45–47, 2006.

[5] D. S. Batory. Feature models, grammars, and propositional formulas. In *9th Int'l Software Product Line Conference, SPLC 2005, Rennes, France, September 26-29, 2005*, volume 3714 of *LNCS*, pages 7–20. Springer, 2005.

[6] D. S. Batory. Feature Models, Grammars, and Propositional Formulas. In *Proceedings of the 9th International Conference on Software Product Lines (SPLC 2005)*, pages 7–20, 2005.

[7] J. Bayer, S. Gerard, Ø. Haugen, J. Mansell, B. Møller-Pedersen, J. Oldevik, P. Tessier, J.-P. Thibault, and T. Widen. *Consolidated Product Line Variability Modeling*, pages 195–241. Springer, 2006.

[8] M. Becker. Towards a general model of variability in product families. In *1st Workshop on Software Variability Management, Groningen, Netherlands, February 2003*, 2003.

[9] D. Benavides, P. T. Martín-Arroyo, and A. R. Cortés. Automated reasoning on feature models. In *7th Int'l Conference on Advanced Information Systems Engineering, CAiSE 2005, Porto, Portugal, June 13-17, 2005*, volume 3520 of *LNCS*, pages 491–503. Springer, 2005.

[10] D. Benavides, A. Ruiz-Cortés, P. Trinidad, and S. Segura. A Survey on the Automated Analyses of Feature Models. In *Jornadas de Ingeniería del Software y Bases de Datos (JISBD)*, 2006.

[11] S. Bühne, K. Lauenroth, and K. Pohl. Why is it not sufficient to model requirements variability with feature models. In *Int'l Workshop on Automotive Requirements Engineering, AuRE 2006, Nagoya, Japan, September 11, 2004*. IEEE Computer Society, 2004.

[12] S. Bühne, K. Lauenroth, and K. Pohl. Modelling requirements variability across product lines. In *13th Int'l Conference on Requirements Engineering (RE 2005), 29 August - 2 September 2005, Paris, France*, pages 41–52. IEEE Computer Society, 2005.

[13] S. Bühne, K. Lauenroth, K. Pohl, and M. Weber. Modelling features for multi-criteria product-lines in the automotive industry. In *Workshop on Software Engineering for Automotive Systems (SEAS), Edinburgh, UK 2004, co-located at ICSE 2004*, pages 9–16, 2004.

[14] J. Coplien, D. Hoffman, and D. Weiss. Commonality and variability in software engineering. *IEEE Softw.*, 15(6):37–45, November 1998.

[15] K. Czarnecki and K. Pietroszek. Verifying feature-based model templates against well-formedness OCL constraints. In *5th Int'l Conference on Generative Programming and Component Engineering, GPCE 2006, Portland, Oregon, USA, October 22-26, 2006*, pages 211–220. ACM, 2006.

[16] D. Dhungana, P. Grünbacher, and R. Rabiser. DecisionKing: A flexible and extensible tool for integrated variability modeling. In *1st Int'l Workshop on Variability Modelling of Software-intensive Systems, VaMoS 2007, Limerick, Ireland, January 16-18, 2007*, pages 119–127. Lero, 2007.

[17] H. Gomaa and M. E. Shin. A Multiple-View Meta-modeling Approach for Variability Management in Software Product Lines. In *Software Reuse: Methods, Techniques and Tools: 8th International Conference, ICSR 2004, Madrid, Spain, July 5-9, 2009. Proceedings*. Springer, July 2004.

[18] I. John, J. Lee, and D. Muthig. Separation of variability dimension and development dimension. In *1st Int'l Workshop on Variability Modelling of Software-intensive Systems, VaMoS 2007, Limerick, Ireland, January 16-18, 2007*, pages 45–49. Lero, 2007.

[19] K. Kang, S. Cohen, J. Hess, W. Novak, and S. Peterson. Feature-Oriented Domain Analysis (FODA) Feasibility Study. Technical Report CMU/SEI-90-TR-21, SEI, Nov. 1990.

[20] K. C. Kang, S. Kim, J. Lee, K. Kim, E. Shin, and M. Huh. FORM: A feature-oriented reuse method with domain-specific reference architectures. *Annals of Softw. Eng.*, 5:143–168, 1998.

[21] K. C. Kang, J. Lee, and P. Donohoe. Feature-oriented project line engineering. *IEEE Softw.*, 19(4):58–65, 2002.

[22] K. Lee, K. C. Kang, and J. Lee. Concepts and guidelines of feature modeling for product line software engineering. In *7th Int'l Conference on Software Reuse, ICSR-7, Austin, TX, USA, April 15-19, 2002*, volume 2319 of *LNCS*, pages 62–77. Springer, 2002.

[23] A. Metzger and P. Heymans. Comparing feature diagram examples found in the research literature. Technical report, Univ. of Duisburg-Essen, 2007.

[24] K. Pohl, G. Böckle, and F. van der Linden. *Software Product Line Engineering: Foundations, Principles and Techniques*. Springer, 2005.

[25] M. Riebisch, K. Böllert, D. Streitferdt, and I. Philippow. Extending Feature Diagrams with UML Multiplicities. In *Proceedings of the Sixth Conference on Integrated Design and Process Technology (IDPT 2002)*, Pasadena, CA, June 2002.

[26] K. Schmid. A comprehensive product line scoping approach and its validation. In *22rd Int'l Conference on Software Engineering, ICSE 2002, 19-25 May 2002, Orlando, Florida, USA*, pages 593–603. ACM, 2002.

[27] P.-Y. Schobbens, P. Heymans, J.-C. Trigaux, and Y. Bontemps. Feature Diagrams: A Survey and A Formal Semantics. In *14th Int'l Requirements Engineering Conference, RE'06, September 2006*, pages 139–148, Minneapolis, Minnesota, USA, 2006.

[28] P.-Y. Schobbens, P. Heymans, J.-C. Trigaux, and Y. Bontemps. Generic semantics of feature diagrams. *Computer Networks*, 51:456–479, February 2007.

[29] M. Sinnema, S. Deelstra, J. Nijhuis, and J. Bosch. Covamof: A framework for modeling variability in software product families. In *3rd Int'l Software Product Line Conference, SPLC 2004, Boston, MA, USA, August 30-September 2, 2004*, volume 3154 of *LNCS*, pages 197–213. Springer, 2004.

[30] C. Sinz. Towards an optimal CNF encoding of boolean cardinality constraints. In *Proc. of the 11th Intl. Conf. on Principles and Practice of Constraint Programming (CP 2005)*, pages 827–831, Sitges, Spain, Oct. 2005.

[31] M. Svahnberg, J. van Gurp, and J. Bosch. A taxonomy of variability realization techniques. *Softw. Pract. Exper.*, 35(8):705–754, July 2005.

Appendix A. Algorithms

Input: An OVM Ω
Output: A VFD$^+$ O

$(N_O, P_O, r_O, \lambda_O, DE_O, CE_O, \Phi_O) \leftarrow (\{\rho\}, \emptyset, \rho, \emptyset, \emptyset, \emptyset, \emptyset)$
where ρ is a new fresh root ;

`% Mapping of mandatory VPs`

Let $MVP \leftarrow \{x \mid x \in VP \wedge \neg Opt(x)\}$;
$\lambda_O \leftarrow \lambda_O \cup \{r_O \mapsto card_{\sharp MVP+1}[\sharp MVP + 1..\sharp MVP + 1]\}$;
foreach $vp \in MVP$ **do**
  $P_O \leftarrow P_O \cup \{vp\}$;
  $DE_O \leftarrow DE_O \cup \{(r_O, vp)\}$;
  mapVariants(vp);
end

`% Mapping of optional VPs`

Let $OVP \leftarrow \{x \mid x \in VP \wedge Opt(x)\}$;
$N_O \leftarrow N_O \cup \{u\}$, where u is a new fresh node ;
$\lambda_O \leftarrow \lambda_O \cup \{u \mapsto card_{\sharp OVP}[0..\sharp OVP]\}$;
$DE_O \leftarrow DE_O \cup \{(r_O, u)\}$;
foreach $vp \in OVP$ **do**
  $P_O \leftarrow P_O \cup \{vp\}$;
  $DE_O \leftarrow DE_O \cup \{(u, vp)\}$;
  mapVariants(vp);
end
$N_O \leftarrow N_O \cup P_O$;

`% Mapping of requires and excludes`

foreach $(vvp_1, vvp_2) \in Req$ **do**
  $\Phi_O \leftarrow \Phi_O \cup \{\neg vvp_1 \vee vvp_2\}$;
end
foreach $(vvp_1, vvp_2) \in Excl$ **do**
  $\Phi_O \leftarrow \Phi_O \cup \{\neg vvp_1 \vee \neg vvp_2\}$;
end

Algorithm 1: Transforming an OVM to a VFD$^+$

Input: An OVM Ω, a partial VFD$^+$ O, and a VP vp in Ω
Output: O gets added the mapping of vp's variants

Let $myVG \leftarrow \{x \mid x \in VG \wedge Parent(x) = vp\}$;
$\lambda_O \leftarrow \lambda_O \cup \{vp \mapsto card_{\sharp myVG}[\sharp myVG..\sharp myVG]\}$;
foreach $vg \in myVG$ **do**
  $N_O \leftarrow N_O \cup \{vg\}$;
  $\lambda_O \leftarrow \lambda_O \cup \{vg \mapsto card_{\sharp vg}[Min(vg)..\ Max'(vg)]\}$[a];
  $DE_O \leftarrow DE_O \cup \{(vp, vg)\}$;
  foreach $v \in vg$ **do**
    $P_O \leftarrow P_O \cup \{v\}$;
    $DE_O \leftarrow DE_O \cup \{(vg, v)\}$;
    $\lambda_O \leftarrow \lambda_O \cup \{v \mapsto card_0[0..0]\}$;
  end
end

Algorithm 2: mapVariants($vp : VP$) subroutine

[a]Instead of Max, we use $Max' : VG \rightarrow \mathbb{N} \triangleq \forall vg \in VG \cdot Max'(vg) = \#vg$ if $Max(vg) = *$, and $Max'(vg) = Max(vg)$ otherwise. This is because OVM's unbounded cardinality (*) only has a meaning for OVM model *evolution* (e.g. when one adds a V to a $0..*$ VG), but not for its 'static' semantics.

Input: Two VFD$^+$, O and F, and X-links χ
Output: A VFD$^+$ G "merging" all the above

$r_G \leftarrow$ a newly created root ;
$N_G \leftarrow N_O \cup N_F \cup \{r_G\}$;
$P_G \leftarrow P_O \cup P_F$;
$\lambda_G \leftarrow \lambda_O \cup \lambda_F \cup \{r_G \mapsto card_2[2..2]\}$;
$DE_G \leftarrow DE_O \cup DE_F \cup \{(r_G, r_O), (r_G, r_F)\}$;
$CE_G \leftarrow \emptyset$;
$\Phi_G \leftarrow \Phi_O \cup \Phi_F$;

`% Mapping of x-links`

foreach $x \in \chi$ **do**
  $\Phi_G \leftarrow \Phi_G \cup \{x\}$
end

Algorithm 3: Creation of a global VFD$^+$